Тестирование компонентов **Vue.js** с помощью **Jest**

Алекс Джовер

@alexjoverm

Тестирование компонентов Vue.js с помощью Jest

Краткое руководство по тестированию компонентов Vue.js с использованием Jest и официальной библиотеки Vue Test Utils.

Alex Jover Morales и Alexey Pyltsyn

Эта книга предназначена для продажи на http://leanpub.com/testingvueru

Эта версия была опубликована на 2018-09-01

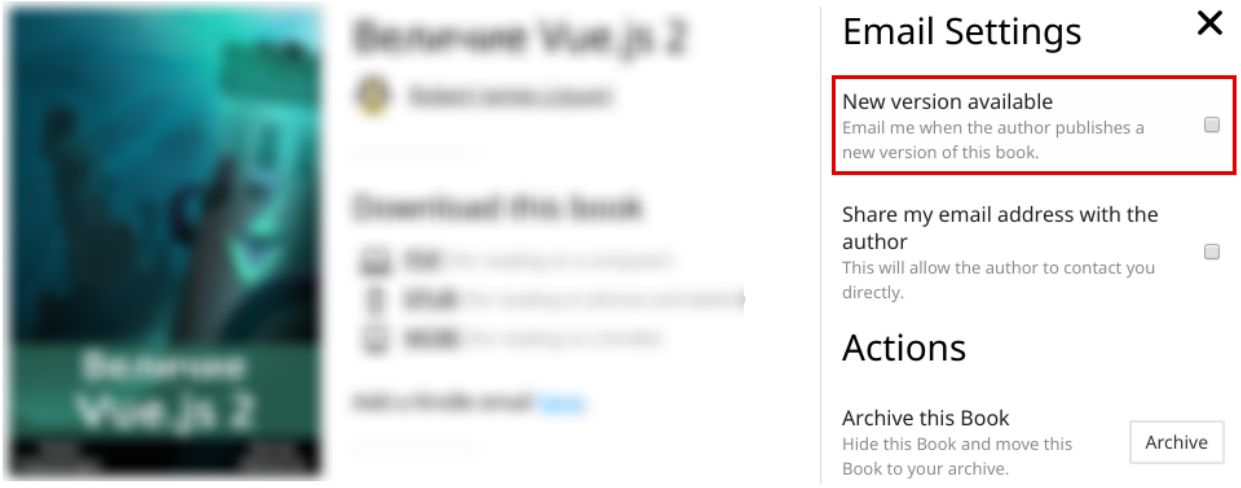
Оглавление

Предисловие

Спасибо, что купили перевод книги Testing Vue.js components with Jest[1]! Если вы купили эту книгу, значит у вас достаточный уровень знаний Vue.js, если это не так, либо вы хотите улучшить свои знания или просто поддержать русское сообщество Vue.js (поскольку я участвую в переводе официальной документации и стараюсь делать это хорошо), то можете рассмотреть покупку переводной книги Величие Vue.js 2[2]!

Обновления

При покупке книги я настоятельно рекомендовал вам подписаться на обновления по электронной почте. Если вы этого не сделали, то есть возможность сделать это сейчас. Зайдите на страницу со всеми вашими книгами на Leanpub[3], выберите эту книгу и справа в блоке «Email Settings» отметьте галочкой «New version available», как показано на скриншоте:



Теперь вы будете знать об обновлениях книги, потому что я планирую обновлять материал, если он будет с течением времени устаревать. Но даже не это главное: я планирую расширять книгу, добавив главы про тестирование Vuex и Vue Router, возможно даже что-то ещё. Так что, держите руку на пульсе! Я не обещаю, что это произойдёт очень быстро, или что даже, это будет в рамках этой книги, но в любом случае вы получите эту информацию **бесплатно**.

[1] https://leanpub.com/testingvuejscomponentswithjest
[2] https://leanpub.com/vuejs2-russian
[3] https://leanpub.com/user_dashboard/library

Принятые соглашения

В тексте данного перевода книги используется буква «ё» и терминология из замечательного словаря «Веб-стандартов»[4], поэтому, в случае непонятного слова, обращайтесь к этому словарю за разъяснением.

Обратная связь

Хоть это и небольшая книга, но ошибки не исключены, несмотря на то, что я попытался сделать всё возможное, чтобы их не было. Однако, если вы нашли опечатку или неточность, пожалуйста, сообщите об этом, создав ишью в репозитории книги[5].

[4] https://github.com/web-standards-ru/dictionary
[5] https://github.com/alexjoverm/testing-vue-book-ru

Что такое тестирование и почему мы должны это делать?

Перевод статьи Alex Jover Morales[6]: What's testing and why should we do it?[7].

Тестирование в области разработки программного обеспечения — это процесс оценки того, что все части приложения ведут себя так, как ожидалось.

Тесты выполняют проверки программного обеспечения, проверяя, что полученный результат соответствует спецификациям, учитывая разные входные данные. Каждая из этих проверок называется *тестовым сценариям* (test case). В рабочем процессе agile каждая пользовательская история должна иметь набор тестовых сценариев. Простой пример:

> US-01: создание компонента ввода валюты
>
> Сценарии тестирования:
>
> – TC-01: он должен принимать только числа
>
> 1. Введите число «2». Ожидается: все должно быть в порядке
> 2. Введите символ «a». Ожидается: должно отображаться сообщение «Разрешены только числа»
>
> – TC02: оно не может быть пустым
>
> …

В тестовых сценариях проверяются требования и характеристики конкретной функциональной возможности (функционала). Они могут предоставлять определённые детали или шаги, чтобы их можно было воспроизвести.

Скорее всего, если вы читаете эту статью, вы знаете, что такое тестирование и TDD, но если вы новичок в тестировании, у вас может возникнуть вопрос, который был в своё время у всех нас:

Не занимает ли тестирование слишком много времени? Нужно ли мне тестировать всё?

В этой статье мы поговорим о плюсах и минусах тестирования, о типах тестирования, и что ещё есть кроме тестирования для обеспечения качества приложения.

[6] *https://github.com/alexjoverm*
[7] *https://vueschool.io/articles/vuejs-tutorials/what-is-testing-and-why-should-we-do-it/*

Зачем писать тесты?

Говоря о том, зачем нужно тестирование, можно упомянуть следующие его преимущества:

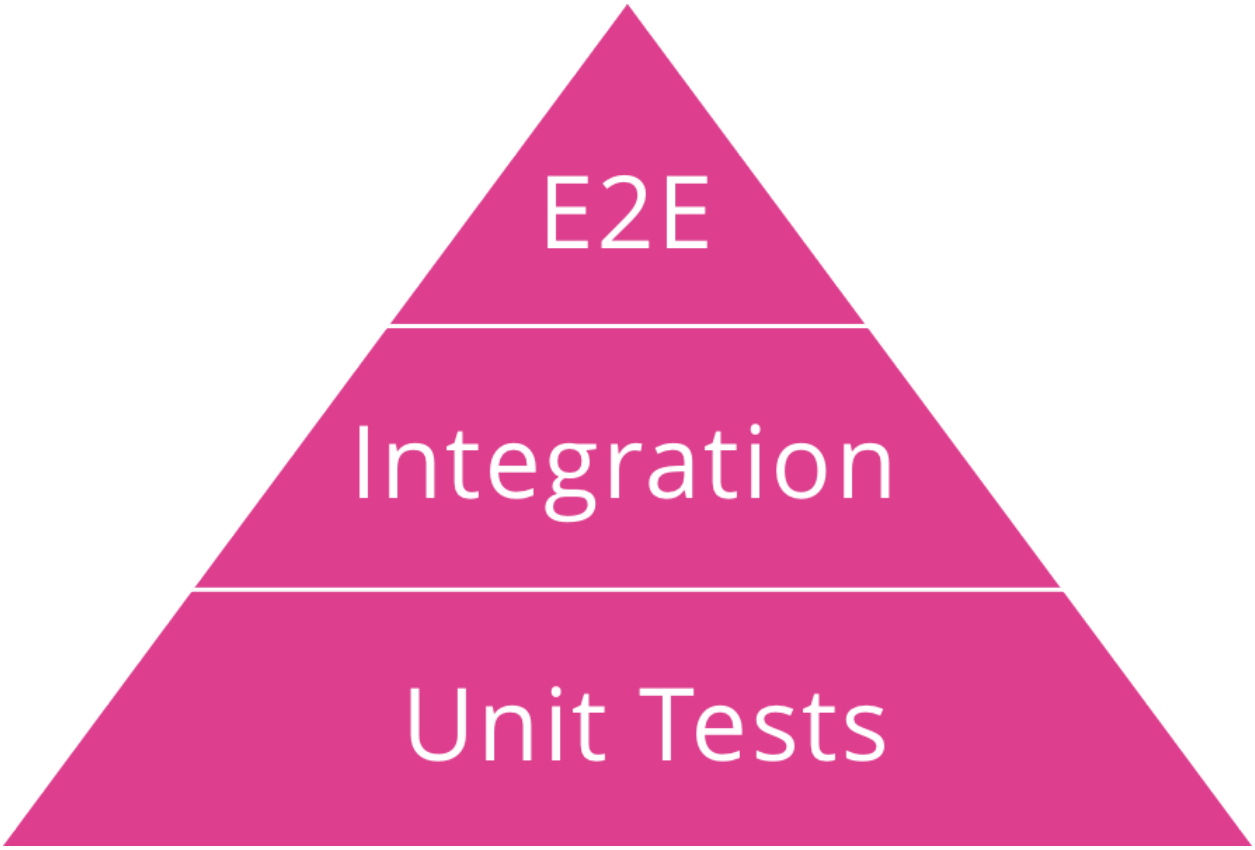
- **Экономит деньги**: без надлежащего тестирования количество времени и ресурсов, необходимых для поддержания продукта в долгосрочной перспективе, намного больше, чем инвестиции в тестирование, не говоря уже о том, что со временем что-то сломается.
- Обеспечивает **безопасность кода при командной работе**: приложение часто создаётся командами. Разные люди изменяют один и тот же фрагмент кода с течением времени. Наличие тестов делают этот процесс более безопасным, поскольку никто не сломает что-то, не узнав об этом. Это также относится и к будущему, тесты обеспечивает безопасность кода, когда вы вернётесь через год или два для внесения изменений.
- Помогает в создании **лучшей архитектуры**: когда часть приложения трудно тестировать, это обычно происходит из-за того, что оно тесно связано с другими частями или функциональность вашего приложения слишком сложна. При их тестировании вам нужно будет сделать их слабосвязанными, применить делегирование и паттерны проектирования, чтобы сделать приложение максимально простым и тестируемым.
- Улучшает **качество кода**: ваш продукт менее подвержен сбоям в работе поскольку тесты помогает написать более надёжный и хороший код, который менее подвержен ошибкам.
- Делает **рефакторинг простым и безопасным**: создание программного обеспечения — это итеративный процесс. Требования меняются с течением времени, следовательно, меняется и функциональность. Наличие хорошего тестового покрытия позволяет вам модифицировать определённый код, проверяя, что тесты все ещё успешно проходят. Если это не так, вы делаете правки в код таким образом, чтобы тесты прошли.

Надеюсь, что на данном этапе вы убедились, почему тестирование полезно для вас, вашего приложения и компании, в которой вы работаете.

Типы тестов

Тесты делятся на разные типы. Каждый тип теста имеет свою цель (назначение) и область действия, и вам следует знать об этом. Каждый разработчик в какой-то момент пишет тест, который тестирует то, чего он не должен.

У нас есть три основных типа тестов:

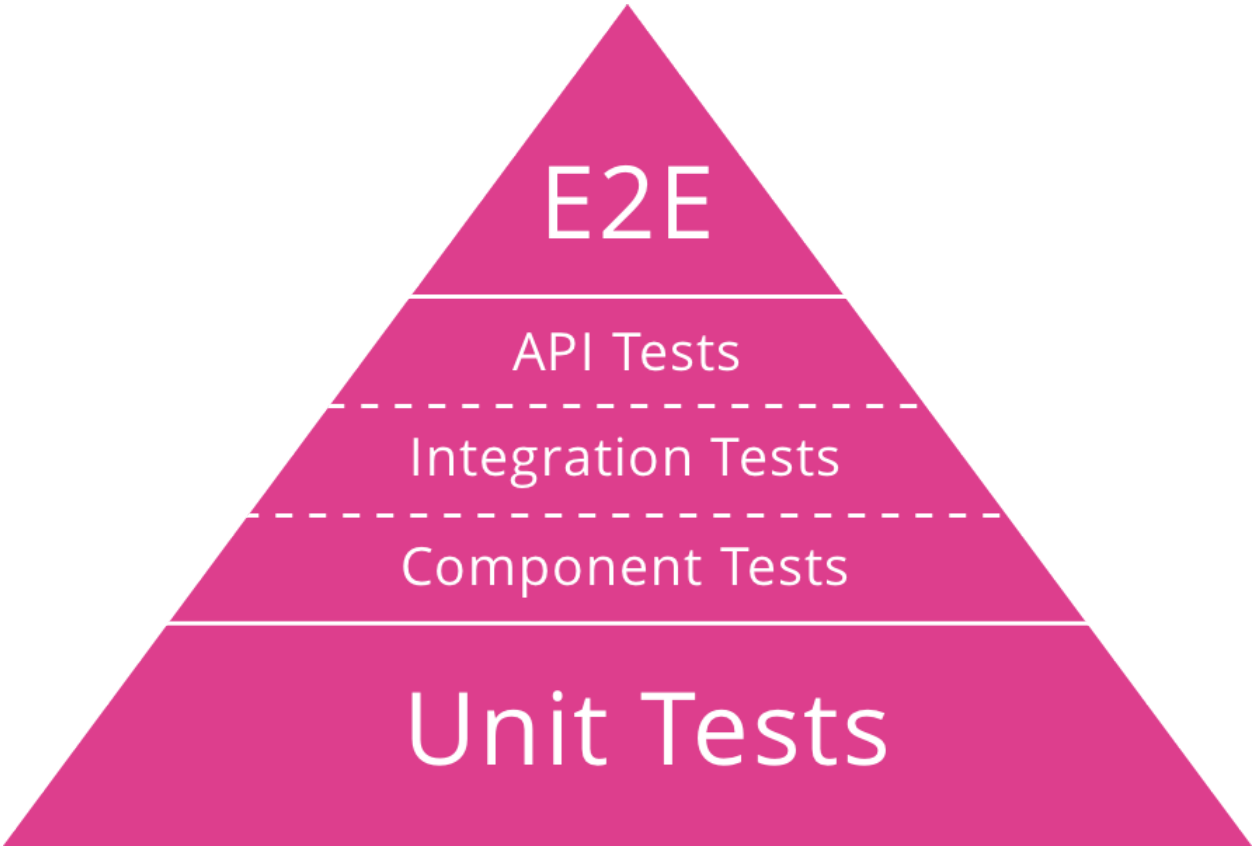


Пирамида тестирования

- **End-to-end-тесты** (сокращённо — E2E) или сквозные тесты, как их называют при переводе: тестируют систему в целом, эмулируя реальную пользовательскую среду. В интернете это тесты, запущенные в браузере, имитирующие щелчки мышью и нажатия клавиш. В общем, набор таких тестов не должен быть большим, так как они дороги для обслуживания и могут легко устареть из-за тестирования системы в целом.
- **Интеграционные тесты**: цель данных тестов состоит в том, чтобы убедиться, что несколько зависимых друг от друга модулей кода работают вместе, и их взаимодействие работает должным образом.
- **Модульные тесты**: они тестируют определённую функциональность изолированно. Их легче всего создавать и обслуживать, поэтому большинство тестовых наборов — это модульные тесты.

Пирамида описывает баланс различных типов тестов. Нижняя часть — это самые быстрые, простые и самые изолированные тесты, а верхние — самые дорогие, самые медленные и охватывают всё приложение в целом.

Интеграционный слой можно даже разделить на ещё большее количество слоёв:



Пирамида тестирования с детально показанным интеграционным слоем

Хотя есть несколько разногласий по поводу количества типов тестов и их имён, наиболее распространёнными являются тесты компонентов и API. Это всего лишь особые типы интеграционных тестов. В частности, тесты компонентов — это тесты, которые мы пишем на стороне фронтенда при тестировании приложения на Vue.js.

Статический анализ

Тесты — не единственный инструмент для обеспечения качества кода. В наше время у JavaScript также есть статическая типизация и инструменты для проверки кода (*linters*, далее — линтеры). Они выполняют статический анализ вашего кода для поиска несоответствий выбранному стилю кода, неправильного использования языка, ненадлежащей и плохой практики, ошибок в контракте данных и многое другое.

Контракт данных — формат данных, который будет использоваться некоторой частью приложения, например функцией. Обычно под этим понимается в каком виде будут представлены данные, например, тип входных и возвращаемых данных.

Статическая типизация делает ваш код более безопасным на основе каждого контракта.

Такие инструменты, как TypeScript или Flow, позволяют определять переменные, параметры и типы возвращаемых значений. Они гарантируют, что ваши классы, функции и методы имеют определённую структуру, а остальная часть вашего кода хорошо работает в соответствии с этим. Многие компании, принявшие статическую типизацию, сразу же поймали несколько ошибок.

Давайте сравним типизированную версию и не типизированную версию функции `sum`:

```
// Нетипизированная функция
function sum(a, b) {
  return a + b;
}

// Типизированная функция
function sum(a: number, b: number) : number {
  return a + b;
}
```

Версия функции без указания типов не мешает нам вызывает её со строчными входными данными, в результате возвращая нам конкатенацию строк. Например, вызов `sum('1', '2')` возвращает `'12'`. Однако, если мы используем статическую типизацию, мы получаем ошибку, такую как `Argument of type '"1"' is not assignable to parameter of type 'number'` (*Аргумент этого типа "'1"' не может быть присвоен параметру типа 'number'*), и эта ошибка типизации предотвращает нас во время компиляции совершить вызов функции, который вернёт неверный результат, поскольку мы ожидаем сложение чисел, а не конкатенацию строк.

Линтеры — это специальные программы, цель которых анализ и проверка различных аспектов кода во время компиляции. JavaScript не имеет преимуществ компилятора, поэтому подвержен ошибкам во время выполнения по сравнению с другими языками, где об ошибках будет сообщено на стадии компиляции. ESLint стал линетром де-факто в JavaScript, а TSLint — в сообществе TypeScript.

Линтер пытается заполнить пробел, предоставляя правила проверки ошибок синтаксиса, стиля кода и неправильного использования (проблемных паттернов). В результате он уменьшает количество ошибок и повышает качество и корректность вашего кода.

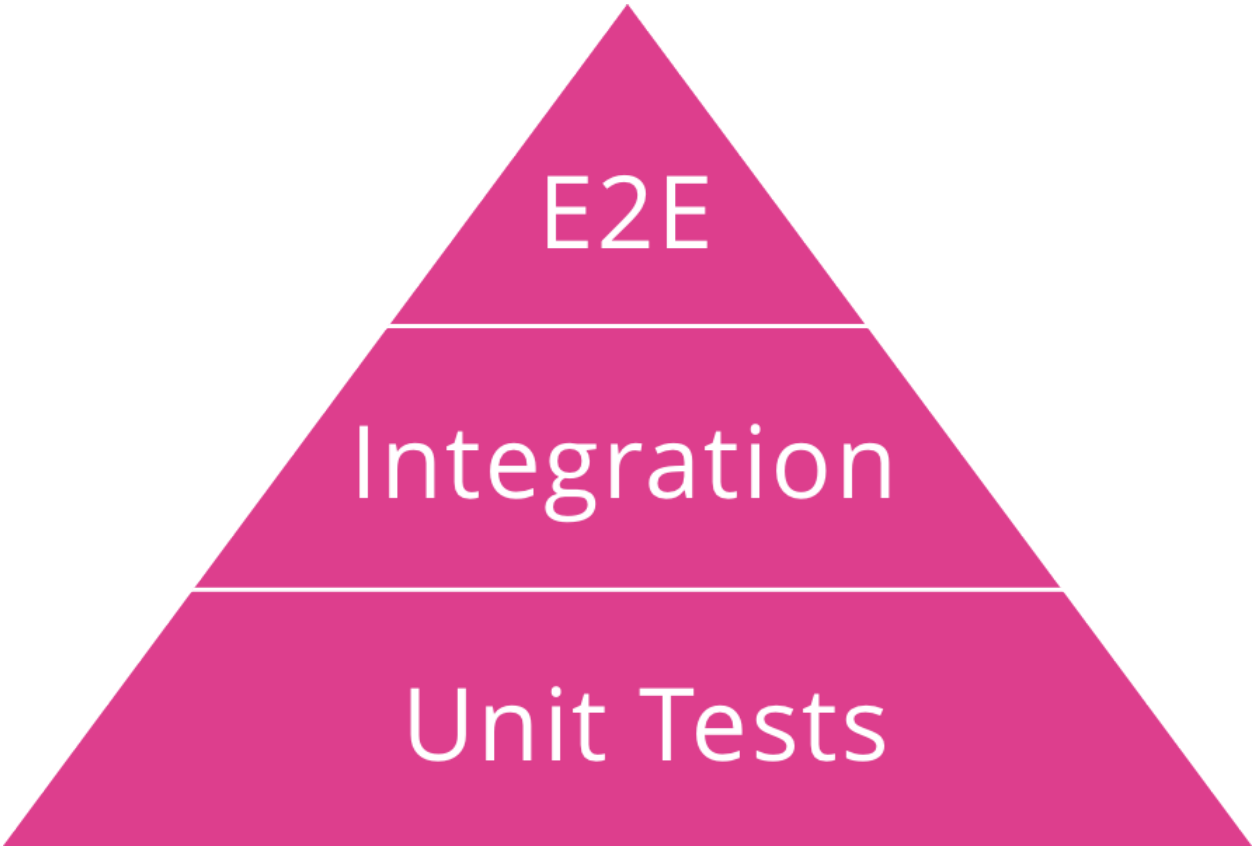
В качестве примера, если вы используете правило no-var[8] в ESLint и напишите следующий код:

```
var greet = 'Эй, Китти';
```

Вы получите ошибку `Unexpected var, используйте let или const вместо (no-var)` (*Неожиданный var, используйте let или const вместо него (no-var)*).

[8] https://eslint.org/docs/rules/no-var

В пирамиде статический анализ ещё точнее определён и проверяется быстрее (почти в режиме реального времени), чем модульные тесты, что делает его основой пирамиды:



Итоговая пирамида тестирования, включающая статический анализ и линтеры

Эта статья[9] является первой частью серии Тестируй как профи в JavaScript[10]. В рамках этой книги она представлена, чтобы дать вам общее представление о тестировании в целом.

[9] https://vueschool.io/articles/vuejs-tutorials/what-is-testing-and-why-should-we-do-it/
[10] https://vueschool.io/articles/series/testing-like-a-pro-in-javascript/

Написание первого модульного теста компонента Vue.js

Узнайте, как писать модульные тесты с помощью официальных инструментов Vue.js и фреймворка Jest.

Vue Test Utils[11], официальная библиотека тестирования VueJS, основанная на avoriaz[12], уже не за горами. @EddYerburgh[13] действительно делает очень хорошую работу, создавая его. Она предоставляет все необходимые инструменты для лёгкого написания модульного теста в приложении, сделанном на VueJS.

Jest[14], с другой стороны, представляет собой фреймворк для тестирования, разработанный в Facebook, позволяющий очень быстро тестировать с потрясающими такими возможностями как:

- Почти нет настроек по умолчанию
- Очень классный интерактивный режим
- Возможность выполнения тестов параллельно
- Шпионы (spies), заглушки (stubs) и подставные объекты (mocks) из коробки
- Встроенное покрытие кода
- Тестирование снимками
- Утилиты имитации модулей

Возможно, вы уже написали тест без этих инструментов, и просто используя karma + mocha + chai + sinon + ..., но вы увидите, насколько проще это может быть.

Настройка примера проекта vue-test

Начнём с создания нового проекта с использованием `vue-cli`[15], отвечая NO на все вопросы с вариантом yes или no:

[11] https://github.com/vuejs/vue-test-utils
[12] https://github.com/eddyerburgh/avoriaz
[13] https://twitter.com/EddYerburgh
[14] https://facebook.github.io/jest
[15] https://github.com/vuejs/vue-cli

```
npm install -g vue-cli
vue init webpack vue-test
cd vue-test
```

Затем нам нужно будет установить кое-какие зависимости:

```
# Установка зависимостей
npm i -D jest jest-vue-preprocessor babel-jest
```

`jest-vue-preprocessor`[16] необходим, чтобы Jest понимал файлы с расширением `.vue`, а `babel-jest`[17] нужен для интеграции с Babel.

Согласно Vue Test Utils, он ~~ещё не выпущен, но теперь вы можете добавить его в свой `package.json` из источника~~:

Обновление (10.10.2017): он может быть установлен уже из npm, так как версия `beta.1` была опубликована.

На заметку

Vue Test Utils[18] предоставляет набор утилит для утверждений (выполнения проверок) на компонентах Vue.js. В книге мы используем последнюю на момент написания версию этого инструмента — `1.0.0-beta.24`.

```
npm i -D vue-test-utils
```

Давайте добавим следующую конфигурацию для Jest в `package.json`:

```
{
  "jest": {
    "moduleNameMapper": {
      "^vue$": "vue/dist/vue.common.js"
    },
    "moduleFileExtensions": ["js", "vue"],
    "transform": {
      "^.+\\.js$": "<rootDir>/node_modules/babel-jest",
      ".*\\.(vue)$": "<rootDir>/node_modules/vue-jest"
    }
  }
}
```

[16] https://github.com/vire/jest-vue-preprocessor
[17] https://github.com/babel/babel-jest
[18] https://github.com/vuejs/vue-test-utils

`moduleFileExtensions` указывает Jest, какие расширения искать, а `transform` — какой препроцессор использовать для расширения файла.

Наконец, добавьте скрипт `test` в `package.json`:

```
{
  "scripts": {
    "test": "jest"
  }
}
```

Тестирование компонента

Далее я буду использовать однофайловые компоненты, и я не проверял, будет ли они работать, если компонент разделить на собственные файлы `html`, `css` или `js`, поэтому давайте предположим, что вы точно также делаете.

Сначала создайте компонент `MessageList.vue` в каталоге `src/components`:

```
<template>
  <ul>
    <li v-for="message in messages">
      {{ message }}
    </li>
  </ul>
</template>

<script>
export default {
  name: 'list',
  props: ['messages']
};
</script>
```

И обновите `App.vue`, чтобы использовать его, следующим образом:

```
<template>
  <div id="app">
    <MessageList :messages="messages"/>
  </div>
</template>

<script>
import MessageList from './components/MessageList'

export default {
  name: 'app',
  data: () => ({ messages: ['Привет, Джон', 'Как дела, Пако?'] }),
  components: {
    MessageList
  }
};
</script>
```

У нас уже есть пара компонентов, которые мы можем протестировать. Давайте создадим каталог `test` в корне проекта и файл `App.test.js`:

```
import Vue from 'vue';
import App from '../src/App';

describe('App.test.js', () => {
  let cmp, vm;

  beforeEach(() => {
    cmp = Vue.extend(App) // Создать копию исходного компонента
    vm = new cmp({
      data: { // Заменить значение данных на эти поддельные данные
        messages: ['Cat']
      }
    }).$mount(); // Создать экземпляр и примонтировать компонент
  })

  it('сообщения идентичны ["Cat"]', () => {
    expect(vm.messages).toEqual(['Cat']);
  });
})
```

На заметку

Обычно сообщения в функции `it` пишутся на английском языке, но сейчас и далее они будут переведены на русский язык.

В данный момент, если мы выполним `npm test` (или `npm t` как сокращённая версия этой команды), тест должен запуститься и успешно пройти. Поскольку мы изменяем тесты, давайте лучше запустим их **в режиме просмотра (watch mode)**:

```
npm t -- --watch
```

Проблема с вложенными компонентами

Этот тест слишком прост. Давайте также проверим, что вывод соответствует ожидаемому. Для этого мы можем использовать удивительную возможность снимков (snapshots) Jest, которая будет генерировать снимок вывода и проводить сравнение с ним в предстоящих запусках. добавьте после предыдущего `it` в `App.test.js`:

```
1  it('имеет ожидаемую структуру HTML', () => {
2    expect(vm.$el).toMatchSnapshot();
3  });
```

Это создаст файл `test/__snapshots__/App.test.js.snap`. Давайте откроем и изучим его:

```
1  // Jest Snapshot v1, https://goo.gl/fbAQLP
2
3  exports[`App.test.js имеет ожидаемую структуру HTML 1`] = `
4  <div
5    id="app"
6  >
7    <ul>
8      <li>
9        Cat
10     </li>
11   </ul>
12 </div>
13 `;
```

Если вы не заметили, здесь есть большая проблема: компонент `MessageList` был также отрисован. **Модульные тесты должны быть протестированы как независимые единицы**, а

это значит, что в `App.test.js` мы хотим протестировать компонент `App` и больше ни какой другой.

Это может стать причиной нескольких проблем. Представьте себе, например, что дочерние компоненты (`MessageList` в этом случае) выполняют операции с побочными эффектами на хуке `created`, такие как вызов `fetch`, действие Vuex или изменения состояния? Это то, чего мы определённо не хотим.

К счастью, **поверхностная или неглубокая отрисовка (Shallow Rendering)** хорошо справляется с этим.

Что такое поверхностная отрисовка?

Поверхностная отрисовка[19] — это метод, который гарантирует, что ваш компонент выполняет отрисовку без дочерних элементов. Это полезно для:

- Тестирование только компонента, который вы хотите протестировать (это как раз то, что означает модульный тест)
- Избегание побочных эффектов, которые могу быть у дочерних компонентов, например, создание HTTP-вызовов, вызовы действий хранилища...

Тестирование компонента с помощью Vue Test Utils

Vue Test Utils предоставим нам поверхностную отрисовку среди прочего функционала. Мы могли бы переписать предыдущий тест следующим образом:

```
1  import { shallowMount } from '@vue/test-utils';
2  import App from '../src/App';
3
4  describe('App.test.js', () => {
5    let cmp;
6
7    beforeEach(() => {
8      cmp = shallowMount(App, { // Создать поверхностный экземпляр компонента
9        data: {
10         messages: ['Cat']
11       }
12     });
13   });
```

[19] http://airbnb.io/enzyme/docs/api/shallow.html

```
14
15    it('сообщения идентичны ["Cat"]', () => {
16      // Внутри cmp.vm, мы имеем доступ ко всем методам экземпляра Vue
17      expect(cmp.vm.messages).toEqual(['Cat']);
18    });
19
20    it('имеет ожидаемую структуру HTML', () => {
21      expect(cmp.element).toMatchSnapshot();
22    });
23  });
```

И теперь, если вы все ещё используете Jest в режиме просмотра, вы увидите, что тест все ещё проходит, но снимок не соответствует. Нажмите `u`, чтобы пересоздать его. Откройте и проверьте его снова:

```
1  // Jest Snapshot v1, https://goo.gl/fbAQLP
2
3  exports[`App.test.js имеет ожидаемую структуру HTML 1`] = `
4  <div
5    id="app"
6  >
7    <!--  -->
8  </div>
9  `;
```

Вы видите? Теперь дочерние элементы не отрисовались, и мы протестировали компонент `App` **полностью изолированным** от дерева компонентов. Кроме того, если у вас есть какие-либо хуки, например `created`, в дочерних компонентах, ни один из них не был вызван ☒.

Если вам интересно, **как реализуется поверхностная отрисовка**, посмотрите исходный код[20], и вы увидите, что в основном создаются заглушки для ключа `components`, метода `render` и хуки жизненного цикла.

В том же духе вы можете реализовать тест `MessageList.test.js`, как представлено ниже:

[20] https://github.com/vuejs/vue-test-utils/blob/dev/packages/shared/stub-components.js

```
import { mount } from '@vue/test-utils'
import MessageList from '../src/components/MessageList'

describe('MessageList.test.js', () => {
  let cmp;

  beforeEach(() => {
    cmp = mount(MessageList, {
      // Помните, что входные параметры переопределяются с помощью `propsData`
      propsData: {
        messages: ['Cat']
      }
    });
  });

  it('получен массив ["Cat"] во входном параметре сообщения', () => {
    expect(cmp.vm.messages).toEqual(['Cat']);
  });

  it('имеет ожидаемую структуру HTML', () => {
    expect(cmp.element).toMatchSnapshot();
  });
});
```

Полную версию примера вы можете найти в репозитории на GitHub[21].

[21] https://github.com/alexjoverm/vue-testing-series/tree/lesson-1

Тестирование компонентов с глубокой вложенностью

Давайте посмотрим, как использовать Vue Test Utils для тестирования полностью отрисованного дерева компонента.

В первой главе мы узнали, как использовать поверхностную отрисовку для тестирования компонента изолированно, предотвращая отрисовку поддерева компонента, т.е. его дочерних элементов.

Но в некоторых случаях мы могли бы протестировать компоненты, которые ведут себя как группа, или молекулы[22], как указано в книге Atomic Design. Имейте в виду, что это относится к презентационным компонентам[23], поскольку они не знают о состоянии и логике приложения. В большинстве случаев вы хотите использовать поверхностную отрисовку для компонентов-контейнеров.

Добавление компонента Message

В случае компонентов Message и MessageList, помимо написания для них модульных тестов, мы могли бы также протестировать их как единое целое.

Начнём с создания файла `components/Message.vue`, в котором определим компонент Message:

```
<template>
  <li class="message">{{message}}</li>
</template>

<script>
  export default {
    props: ['message']
  }
</script>
```

И обновим `components/MessageList.vue` для его использования:

[22] http://atomicdesign.bradfrost.com/chapter-2/#molecules
[23] https://medium.com/@dan_abramov/smart-and-dumb-components-7ca2f9a7c7d0

```
<template>
  <ul>
    <Message :message="message" v-for="message in messages"/>
  </ul>
</template>

<script>
import Message from './Message'

export default {
  props: ['messages'],
  components: {
    Message
  }
}
</script>
```

Тестирование `MessageList` с компонентом `Message`

Для тестирования MessageList с полной (глубокой) отрисовкой, нам нужно просто использовать `mount` вместо `shallow` в ранее созданном тесте в файле `test/MessageList.test.js`:

```
import { mount } from '@vue/test-utils'
import MessageList from '../src/components/MessageList'

describe('MessageList.test.js', () => {
  let cmp;

  beforeEach(() => {
    cmp = mount(MessageList, {
      // Помните, что входные параметры переопределяется с помощью `propsData`
      propsData: {
        messages: ['Кот']
      }
    });
  });

  it('получен массив ["Кот"] в качестве входного параметра message', () => {
    expect(cmp.vm.messages).toEqual(['Кот'])
  });

```

```
  it('имеет ожидаемую структуру HTML', () => {
    expect(cmp.element).toMatchSnapshot()
  });
});
```

Кстати говоря, вы поняли, что такое `beforeEach`? Это очень элегантный способ создания нового компонента перед каждым выполнением теста, что очень важно при модульном тестировании, поскольку он определяет, что тест не должен зависеть друг от друга.

Как `mount`, так и `shallow` используют точно такой же API, разница только в отрисовке. Я покажу вам постепенно API далее в этой серии.

Если вы запустите `npm t`, то увидите, что тест завершился неудачей, потому что снимок не соответствует `MessageList.test.js`. Чтобы пересоздать его, запустите выполнение тестов с помощью опции `-u`:

```
npm t -- -u
```

Затем, если вы откроете и посмотрите содержимое `test/__snapshots__/MessageList.test.js.snap`, то увидите присутствие `class="message"`, что означает, что компонент отрисован в соответствии с последними изменениями.

```
// Jest Snapshot v1, https://goo.gl/fbAQLP

exports[`MessageList.test.js имеет ожидаемую структуру HTML 1`] = `
<ul>
  <li
    class="message"
  >
    Кот
  </li>
</ul>
`;
```

Помните о том, чтобы **избегать использование глубокой отрисовки в случаях, когда могут быть побочные эффекты**, поскольку хуки компонентов дочерних элементов, такие как `created` и `mount`, будут запускаться, и там могут быть HTTP-вызовы или другие операции, которые таким образом будут выполнены, когда как мы при тестировании мы этого не хотим. Если вы хотите попробовать в действии то, о чем я только что написал, добавьте в компонент `Message.vue` вызов `console.log` в хуке `created`:

```
export default {
  props: ['message'],
  created() {
    console.log('СОЗДАН!');
  }
};
```

Теперь, если вы снова запустите тесты с помощью `npm t`, увидите текст `"СОЗДАН!"` в выводе терминала. Поэтому будьте осторожны.

Вы можете найти полный пример на GitHub[24].

[24] https://github.com/alexjoverm/vue-testing-series/tree/https://github.com/alexjoverm/vue-testing-series/tree/Test-fully-rendered-Vue-js-Components-in-Jest

Тестирование стилей и структуры компонентов Vue.js

Пока что в тестах мы использовали снимки Jest[25]. Это здорово, но иногда мы хотим проверить (или утверждать что-либо) что-то более конкретное.

Хотя вы можете получить доступ к экземпляру Vue через `cmp.vm`[26], в вашем распоряжении есть набор утилит, чтобы получить его проще. Давайте посмотрим, что мы можем сделать.

Объект Wrapper

`Wrapper` — главный объект Vue Test Utils. Это тип, возвращаемый функциями `mount`, `shallow`, `find` и `findAll`. Вы можете посмотреть здесь[27] весь API и типы.

Методы `find` и `findAll`

Они принимают параметр Selector[28] в качестве аргумента, который может быть как CSS-селектором, так и Vue-компонентом.

Поэтому мы можем сделать что-то подобное:

```
1  const messageListCmp = mount(MessageList);
2
3  expect(messageListCmp.find('.message').element).toBeInstanceOf(HTMLElement);
4
5  // Или даже вызывать его несколько раз
6  let el = messageListCmp.find('.message').find('span').element;
7
8  // Хотя предыдущий пример мы могли сделать это короче
9  let el = messageListCmp.find('.message span').element;
```

Утверждение структуры и стиля

Давайте добавим больше тестов для `MessageList.test.js`:

[25] https://facebook.github.io/jest/docs/snapshot-testing.html
[26] https://github.com/alexjoverm/vue-testing-series/blob/master/test/MessageList.test.js#L17
[27] https://github.com/vuejs/vue-test-utils/blob/dev/packages/test-utils/types/index.d.ts#L84
[28] https://github.com/vuejs/vue-test-utils/blob/dev/packages/test-utils/types/index.d.ts#L17

```
it('это компонент MessageList', () => {
  expect(messageListCmp.is(MessageList)).toBe(true);

  // Или с помощью CSS-селектора
  expect(messageListCmp.is('ul')).toBe(true);
});

it('содержит компонент Message', () => {
  expect(cmp.contains(Message)).toBe(true);

  // Или с помощью CSS-селектора
  expect(cmp.contains('.message')).toBe(true);
});
```

Здесь мы используем `is` для утверждения типа корневого компонента и `contains` для проверки существования дочерних компонентов. Так же, как и `find`, они получают объект типа Selector, который может быть CSS-селектором (тип Selector) или компонентом (тип Component).

У нас есть некоторые утилиты для утверждения **экземпляра Vue**:

```
it('Компоненты MessageList и Message являются экземплярами Vue', () => {
  expect(cmp.isVueInstance()).toBe(true);
  expect(cmp.find(Message).isVueInstance()).toBe(true);
});
```

Теперь мы собираемся проверить **структуру** компонента более подробно:

```
it('Существует элемент Message', () => {
  expect(cmp.find('.message').exists()).toBe(true);
});

it('Message не пустой', () => {
  expect(cmp.find(Message).isEmpty()).toBe(false);
});

it('У Message есть атрибут класса со значением "message"', () => {
  expect(cmp.attributes().class).toBe('message');
});
```

Методы `exists`, `isEmpty` очень удобные на практике.

Теперь у нас есть `classes()` и `attributes().style` для проверки **стилей**. Давайте обновим компонент `Message.vue` со стилем, поскольку `attributes().style` проверяет только встроенные стили:

```
<li style="margin-top: 10px" class="message">{{message}}</li>
```

Вот тесты для этого:

```
it('У компонента Message задан класс .message', () => {
  expect(cmp.find(Message).classes()).toContain('message');
});

it('У компонента Message определён стиль `padding-top: 10`', () => {
  expect(cmp.find(Message).attributes().style).toBe('margin-top: 10px;');
});
```

Резюме

Для упрощения тестирования компонентов Vue существует куча утилит. Вы можете найти их все в файле типизации[29].

Вы можете найти рабочий код этой главы в этом репозитории[30].

[29] https://github.com/vuejs/vue-test-utils/blob/dev/packages/test-utils/types/index.d.ts
[30] https://github.com/alexjoverm/vue-testing-series/blob/Test-Styles-and-Structure-in-Vue-js-and-Jest/test/MessageList.test.js

Тестирование свойств и пользовательских событий в компонентах Vue.js

Существуют различные способы тестирования свойств и событий, включая пользовательские.

Свойства — это пользовательские атрибуты, переданные от родительских к дочерним компонентам. С пользовательскими событиями всё устроено как раз наоборот: они отправляют данные непосредственному родителю через событие. Оба они объединяются — это провода взаимодействия и коммуникации в компонентах Vue.js.

В модульном тестировании тестирование входов и выходов (свойств и пользовательских событий) означает тестирование того, как компонент ведёт себя изолированно, когда он получает и отправляет данные. Самое время закатать рукава!

Свойства

Когда мы тестируем свойства компонента, мы можем проверить, как работает компонент, когда мы передаём ему определённые свойства. Но прежде чем продолжить, важно отметить:

Чтобы передать свойства компонентам, используйте `propsData`, а не `props`. Последнее определяет свойства, а не передавать в них данные.

Сначала создайте файл `Message.test.js` и добавьте следующий код:

```
describe('Message.test.js', () => {
  let cmp;

  describe('Свойства', () => {
    // @TODO
  });
});
```

Мы сгруппируем тестовые сценарии в выражении `describe`, которые могут быть вложенными. Поэтому мы можем использовать эту стратегию для группировки тестов свойств и событий по отдельности.

Затем мы создадим вспомогательную фабричную функцию для создания компонента сообщения, предоставим некоторые свойства

```
const createCmp = propsData => mount(Message, { propsData });
```

Тестирование существование свойства

Две очевидные вещи, которые мы можем протестировать — свойство существует или нет. Помните, что у компонента `Message.vue` есть свойство `message`, поэтому давайте утверждать, что оно в действительности получает это свойство. Vue Test Utils поставляется с функцией `props()`, используя её в сочетании с `toBe` мы можем проверить, что определённое свойство имеет заданное значение:

```
it('есть свойство message', () => {
  cmp = createCmp({ message: 'hey' });
  expect(cmp.props().message).toBe('hey');
});
```

Свойства ведут таким образом, потому что что они будут получены, только если объявлены в компоненте. Это означает, что если мы передадим **свойство**, **которое не определено**, **оно не будет получено**. Поэтому, чтобы проверить отсутствие существования свойства, используйте несуществующее свойство:

```
it('входной параметр `cat` не определён', () => {
  cmp = createCmp({ cat: 'hey' });
  expect(cmp.props().cat).toBeUndefined();
});
```

Однако хотя в данном случае тест завершится удачно, не стоит забывать, что у Vue есть обычные атрибуты[31], которые устанавливаются как атрибуты корневого элемента в компоненте `Message`, поэтому можно проверить, что это поведение также работает, убедившись, что обычное свойство (не входной параметр) существует с помощью `attributes()`.

```
it('обычное свойство `cat` существует', () => {
  cmp = createCmp({ cat: 'hey' });
  expect(cmp.attributes().cat).toBe('hey');
});
```

Мы также можем проверить **значение по умолчанию**. Перейдите в `Message.vue` и измените входные параметры следующим образом:

[31] https://vuejs.org/v2/guide/components.html#Non-Prop-Attributes

```
props: {
  message: String,
  author: {
    type: String,
    default: 'Петя'
  }
},
```

Тогда тест будет такой:

```
it('Имя по умолчанию — Петя', () => {
  cmp = createCmp({ message: 'hey' });
  expect(cmp.props().author).toBe('Петя');
})
```

Утверждение проверки свойств

У свойств могут быть заданы правила проверки, гарантирующие, что свойство обязательно или должно иметь определённый тип. Давайте расширим объявления свойства `message` следующим образом:

```
props: {
  message: {
    type: String,
    required: true,
    validator: message => message.length > 1
  }
}
```

Идём дальше, вы можете использовать пользовательские типы конструкторов или настраиваемые правила проверки, как вы указано в документации[32]. Не делайте этого прямо сейчас, я просто показываю это в качестве примера:

[32] https://vuejs.org/v2/guide/components.html#Prop-Validation

```
class Message {}
// ...
props: {
  message: {
    type: Message, // Сравнение происходит с использованием `instance of`
    // ...
    }
  }
}
```

Всякий раз, когда правило проверки не выполняется, Vue показывает `console.error`. Например, для `createCmp({message: 1})` будет отображаться следующая ошибка:

[Vue warn]: Invalid prop: type check failed for prop “message”. Expected String, got Number. (found in <Root>)

На текущий момент Vue Test Utils не имеет никакой утилиты для проверки подобного. Мы могли бы использовать `jest.spyOn` для тестирования ошибки:

```
it('message is of type string', () => {
  let spy = jest.spyOn(console, 'error');
  cmp = createCmp({ message: 1 });
  expect(spy).toBeCalledWith(expect.stringContaining('[Vue warn]: Invalid prop'));

  spy.mockReset(); // Или mockRestore(), чтобы полностью удалить mock-объект
});
```

Здесь мы следим за функцией `console.error` и проверяем, что оно отображает сообщение, содержащее заданную строку. Это не идеальный способ проверки, поскольку мы следим за глобальными объектами и полагаемся на побочные эффекты.

К счастью, есть более простой способ сделать это, проверяя `vm.$options`. В этом свойстве Vue хранит параметры компонента в «расширенном виде». Под этим я имею в виду, что вы можете определить свои свойства по-разному:

```
props: ['message']

// Или так:

props: {
  message: String
}

// Или даже так:

props: {
  message: {
    type: String
  }
}
```

Но как бы вы не определили свойства, все они окажутся в самой расширенной форме объекта (как и последняя). Поэтому, если мы проверим `cmp.vm.$options.props.message`, в первом нами рассматриваемом случае все они будут в формате `{type: X}` (хотя для первого примера из блока кода выше это будет `{ type: null }`, поскольку тип свойства не был задан).

Теперь, зная про всё это, мы могли бы написать набор тестов, проверяющий, что свойство `message` имеет ожидаемые правила проверки:

```
describe('Message.test.js', () => {
  // ...
  describe('Свойства', () => {
    // ...
    describe('Тестирование корректности проверки свойств', () => {
      const message = createCmp().vm.$options.props.message;

      it('Входной параметр message имеет строковый тип', () => {
        expect(message.type).toBe(String);
      });

      it('Входной параметр message является обязательным', () => {
        expect(message.required).toBeTruthy();
      });

      it('Входной параметр message имеет не менее двух символов', () => {
        expect(message.validator && message.validator('a')).toBeFalsy();
        expect(message.validator && message.validator('aa')).toBeTruthy();
      });
```

```
    });
  });
};
```

Пользовательские события

Мы можем проверить как минимум два момента в пользовательских событиях:

- Утверждение, что после действия срабатывает событие
- Проверка того, что обработчик событий работает, когда он срабатывает

В конкретном случае наших компонентов `MessageList.vue` и `Message.vue`, мы можем сделать следующее:

- Проверить, что компоненты `Message` запускают событие `message-clicked` при нажатии на сообщение
- Проверить, что когда в компоненте `MessageList` происходит событие `message-clicked`, то вызывается функция `handleMessageClick`

Сначала перейдите в `Message.vue` и используйте `$emit` для генерирования упомянутого пользовательского события:

```
<template>
  <li
    style="margin-top: 10px"
    class="message"
    @click="handleClick">
      {{message}}
  </li>
</template>

<script>
export default {
  name: 'Message',
  props: ['message'],
  methods: {
    handleClick() {
      this.$emit('message-clicked', this.message)
    }
  }
};
</script>
```

А в компоненте `MessageList.vue` обрабатывает это событие `@message-clicked`:

```
<template>
  <ul>
    <Message
      @message-clicked="handleMessageClick"
      :message="message"
      v-for="message in messages"
      :key="message"/>
  </ul>
</template>

<script>
import Message from './Message'

export default {
  name: 'MessageList',
  props: ['messages'],
  methods: {
    handleMessageClick(message) {
      console.log(message)
    }
  },
  components: {
    Message
  }
};
</script>
```

Теперь пришло время написать модульный тест. Создайте вложенный `describe` в файле `test/Message.spec.js` и подготовьте заглушку для тестового сценария *“Проверить, что компоненты `Message` запускают событие `message-clicked` при нажатии на сообщение”*, про которое было написано ранее:

```
//...
describe('Message.test.js', () => {
  // ...
  describe('События', () => {
    beforeEach(() => {
      cmp = createCmp({ message: 'Cat' });
    });

    it('вызывается handleClick после клика на сообщение', () => {
      // @TODO
    });
  })
})
```

Тестирование события клика вызывает метод-обработчик

Первое, что мы можем проверить, это то, что при нажатии сообщения вызывается функция `handleClick`. Для этого мы можем использовать `trigger` компонента-обёртки и шпион из Jest, используя функцию `spyOn`:

```
it('вызывается handleClick после клика на сообщение', () => {
  const spy = spyOn(cmp.vm, 'handleClick');
  // cmp.update() // Не требуется с vue-test-utils версии 1.0.0-beta.12

  const el = cmp.find('.message').trigger('click');
  expect(cmp.vm.handleClick).toBeCalled();
});
```

Внимание: ранее при изменениях в шаблоне требовался вызов `cmp.update()`, в новых версиях этот метод удалён, поскольку все обновления по умолчанию синхронные, поэтому при его использовании, вы получите следующую ошибку: `[vue-test-utils]: update has been removed from vue-test-utils. All updates are now synchronous by default`

Имейте в виду, что с помощью шпиона будет вызываться изначальный метод `handleClick`. Возможно, вы действительно этого хотите, но, как правило, мы хотим избежать подобного при тестировании, а вместо этого просто хотим проверить, что при клике на самом деле вызываются определённые методы. Для этого мы можем использовать функцию-имитацию, предоставляемую Jest:

```
it('вызывается handleClick при клике на сообщение', () => {
  cmp.vm.handleClick = jest.fn();

  const el = cmp.find('.message').trigger('click');
  expect(cmp.vm.handleClick).toBeCalled();
})
```

Здесь мы полностью заменяем метод `handleClick`, доступный в `vm` компонента-оболочки, возвращаемого функцией `mount`.

Однако выше написанный тест завершится неудачей, поскольку мы не указали, что хотим использовать функцию-имитацию, а не оригинальный метод-обработчик. Для исправления этого воспользуемся вспомогательным методом `setMethods` из официального инструментария:

```
it('вызывается handleClick при клике на сообщение', () => {
  const stub = jest.fn();
  cmp.setMethods({ handleClick: stub });

  const el = cmp.find('.message').trigger('click');
  expect(stub).toBeCalled();
})
```

Использование **`setMethods` — это предлагаемый способ** для проверки вызова обработчика при наступлении события, поскольку это абстракция, который официальный инструмент предоставляет нам для случаев, когда изменяются внутренности Vue.

Тестирование генерируемого пользовательского события `message-clicked`

Мы протестировали, что метод клика вызывает обработчик, но мы не тестировали, что обработчик сам генерирует событие `message-clicked`. Мы можем напрямую вызвать метод `handleClick` и использовать функцию-имитации из Jest в сочетании с методом экземпляра Vue `$on`:

```
it('запускает событие message-clicked при вызове метода handleClick', () => {
  const stub = jest.fn();
  cmp.vm.$on('message-clicked', stub);
  cmp.vm.handleClick();

  expect(stub).toBeCalledWith('Cat');
})
```

Смотрите, что здесь мы используем `toBeCalledWith`, поэтому мы можем точно проверить, какие параметры мы ожидаем, делая тест ещё более надёжным.

Тестирование @message-clicked генерирует событие

Для пользовательских событий мы не можем использовать метод `trigger`, так как он только для DOM-событий. Но мы можем сами сгенерировать событие, получив компонент `Message`, используя для этого метод `vm.$emit`.

Поэтому добавьте следующий тест к `MessageList.test.js`:

```
it('вызывается handleMessageClick при срабатывании @message-click', () => {
  const stub = jest.fn();
  cmp.setMethods({ handleMessageClick: stub });
  const el = cmp.find('.message').vm.$emit('message-clicked', 'Cat');

  expect(stub).toBeCalledWith('Cat');
})
```

Однако мы можем написать тот же самый тест, т.е. проверить вызов `handleMessageClick`, который находится в дочернем компоненте, из файла для тестов к родительскому компоненту `Message`.

Для этого воспользуемся методом `emitted()`, который возвращает массив вызовов событий, поэтому так как в тесте мы генерируем только один вызов события `message-clicked`, то обращаемся в элементу массива с нулевым индексом, а далее проверяем, что было передано в момент порождения события. Можете этого не добавлять, просто знайте, что есть такая возможность.

```
it('вызывается handleMessageClick при срабатывании @message-click', () => {
    // Файл `Message.test.js`:

    const el = cmp.find('.message').vm.$emit('message-clicked', 'Cat');

    expect(cmp.emitted()['message-clicked'][0]).toEqual(['Cat']);
});
```

Резюме

В этой главе увидели несколько случаев проверки свойств и событий. С использованием Vue Test Utils, официального инструмента тестирования Vue, тестировать всё это намного проще.

Вы можете найти рабочий код, который мы использовали здесь в этом репозитории[33].

[33] https://github.com/alexjoverm/vue-testing-series/tree/Test-Properties-and-Custom-Events-in-Vue-js-Components-with-Jest

Тестирование вычисляемых свойств и наблюдателей в компонентах Vue.js

В этой главе вы узнаете о тестировании реактивности вычисляемых свойств и наблюдателей во Vue.js.

Вычисляемые свойства и наблюдатели — реактивные части логики компонентов Vue.js. Они оба предназначены для достижения совершенно других целей, один для синхронной, а другой для асинхронной, что делает их поведение несколько иным.

Вычисляемые свойства

Вычисляемые свойства — это простые реактивные функции, которые возвращают данные в другой форме. Они ведут себя точно так же, как стандартные свойства `get/set` в языке:

```
class X {
  // ...

  get fullName() {
    return `${this.name} ${this.surname}`
  }

  set fullName() {
    // ...
  }
}
```

Фактически, когда вы создаёте компоненты Vue на основе классов, как я объясняю в своём курсе на Egghead «Использование TypeScript для разработки веб-приложений Vue.js»[34] (на английском), вы напишете вычисляемые свойства именно так. Если вы используете обычные объекты, это будет так, как показано ниже:

[34] https://egghead.io/courses/use-typescript-to-develop-vue-js-web-applications

```
export default {
  // ...
  computed: {
    fullName() {
      return `${this.name} ${this.surname}`
    }
  }
};
```

И вы даже можете добавить `set` следующим образом:

```
export default {
  computed: {
    fullName: {
      get() {
        return `${this.name} ${this.surname}`
      },
      set() {
        // ...
      }
    }
  }
};
```

Тестирование вычисляемых свойств

Тестирование вычисляемого свойства очень просто и, возможно, иногда вы тестируете не только конкретно вычисляемое свойство, а как её как часть других тестов. Но в большинстве случаев неплохо иметь тесты для подобных свойств, независимо от того, что вычисляемое свойство очищает входное поле ввода или объединяет данные, поэтому мы хотим убедиться, что все работает должным образом. Итак, давайте начнём.

Прежде всего, создайте компонент `Form.vue`:

```
<template>
  <div>
    <form action="">
      <input type="text" v-model="inputValue">
      <span class="reversed">{{ reversedInput }}</span>
    </form>
  </div>
</template>

<script>
export default {
  props: ['reversed'],
  data: () => ({
    inputValue: ''
  }),
  computed: {
    reversedInput() {
      return this.reversed ?
        this.inputValue.split('').reverse().join('') :
        this.inputValue
    }
  }
};
</script>
```

Он будет показывать введённое в поле ввода значение, а рядом с ним ту же строку, но перевёрнутую наоборот. Это всего лишь простой пример, но это достаточно для проверки вычисляемого свойства.

Теперь сделайте изменения в `App.vue`, импортировав созданный компонент `Form` сразу после `MessageList` и не забудьте включить его в свойство объекта `components`. Затем создайте `test/Form.test.js` со стандартной заглушкой, которую мы уже использовали в других тестах:

```
import { shallowMount } from '@vue/test-utils';
import Form from '../src/components/Form';

describe('Form.test.js', () => {
  let cmp;

  beforeEach(() => {
    cmp = shallowMount(Form);
  });
});
```

Теперь создайте набор тестов с двумя тестовыми сценариями:

```
describe('Свойства', () => {
  it('возвращает строку в обычном порядке, если свойство reversed не равняется true'\
, () => {
    cmp.setData({ inputValue: 'Yoo' });
    expect(cmp.vm.reversedInput).toBe('Yoo');
  });

  it('возвращает перевёрнутую строку, если свойство reversed равняется true', () => {
    cmp.setData({ inputValue: 'Yoo' });
    cmp.setProps({ reversed: true });
    expect(cmp.vm.reversedInput).toBe('ooY');
  });
});
```

Мы можем получить доступ к экземпляру компонента в `cmp.vm`, т.е. к внутреннему свойству, вычисляемым свойствам и методам. Теперь для тестирования, нам просто нужно изменить значение и убедиться, что возвращается одинаковая строка, если входной параметр `reversed` равен `false`.

Второй тестовый сценарий во многом похож на первым, лишь за тем исключением, что мы устанавливаем для свойства `reversed` значение `true`. Мы могли бы использовать `cmp.vm ...` для изменения входного параметра, но Vue Test Utils предоставляет нам вспомогательный метод `setProps({ property: value, ... })`, что довольно просто для использования.

Вот и все, в зависимости от вычисляемого свойства, может потребоваться больше тестовых сценариев.

Наблюдатели

Честно говоря, я не сталкивался ни с одним случаем, когда мне действительно нужно было использовать наблюдателей, где бы вычисляемые свойства не смогли справиться. Я тоже видел как ими злоупотребляют, что приведёт к очень нечёткому потоку данных между компонентами и запутыванию всего, поэтому не спешите использовать их и подумайте заранее.

Как вы можете видеть в документации Vue.js[35], наблюдатели часто используются для реагирования на изменения данных и выполнения асинхронных операций, например, это может быть выполнение AJAX-запросов.

[35] https://ru.vuejs.org/v2/guide/computed.html#%D0%9C%D0%B5%D1%82%D0%BE%D0%B4%D1%8B-%D0%BD%D0%B0%D0%B1%D0%BB%D1%8E%D0%B4%D0%B0%D1%82%D0%B5%D0%BB%D0%B8

Тестирование наблюдателей

Предположим, мы хотим что-то сделать, когда изменится `inputValue` из состояния. Мы могли бы выполнить AJAX-запрос, но поскольку это более сложно, и мы увидим это в следующем уроке, а пока давайте просто напишем `console.log`. Добавьте свойство `watch` в опции компонента `Form.vue`:

```
watch: {
  inputValue(newVal, oldVal) {
    if (newVal.trim().length && newVal !== oldVal) {
      console.log(newVal);
    }
  }
}
```

Обратите внимание, что функция-наблюдатель `inputValue` соответствует имени переменной состояния. По соглашению, Vue будет искать его в состоянии объектов `properties` и `data`, используя имя watch-функции, в этом случае это `inputValue`, который можно найти в `data`, и он добавит наблюдателя туда.

Посмотрите, функция-наблюдатель принимает новое значение в качестве первого параметра, а старый — вторым. В данном случае мы решили логировать только тогда, когда новое значение не пустое, а её значение отличается от старого значения. Обычно мы хотели бы написать тест для каждого случая, в зависимости от времени, сколько у вас тестов и насколько критичен этот код.

Что мы должны протестировать в функции-наблюдателе? Ну, об этом мы ещё поговорим далее в следующем уроке, когда затронем методы тестирования, но предположим, что мы просто хотим знать, что она вызывает `console.log`, когда это нужно. Итак, давайте добавим заготовку для набора тестов наблюдателей в `Form.test.js`:

```
describe('Form.test.js', () => {
  let cmp;

  describe('Наблюдатели - inputValue', () => {
    let spy;

    beforeAll(() => {
      spy = jest.spyOn(console, 'log');
    });

    afterEach(() => {
      spy.mockClear();
    });
```

```
14
15      it('не вызывается, если значение пустое (с удалением пробелов)', () => {
16        // @TODO
17      });
18
19      it('не вызывается, если значение одно и то же', () => {
20        // @TODO
21      });
22
23      it('вызывается с новым значением в других случаях', () => {
24        // @TODO
25      });
26    });
27  });
```

Мы используем шпион для метода `console.log`, инициализируем его перед началом каждого теста и перезапускаем его состояние после каждого из них, чтобы они начинались с нового шпиона.

Для тестирования функции-наблюдателя, нам просто нужно изменить значение на другое, которое установлено для наблюдателя, в данном случае это состояние `inputValue`. Но здесь есть кое-что странное... давайте начнём с последнего теста.

```
1  it('вызывается с новым значением в других случаях', () => {
2    cmp.vm.inputValue = 'foo';
3    expect(spy).toBeCalled();
4  });
```

Мы изменяем значение `inputValue`, поэтому теперь должен вызван шпион `console.log`, правильно? Да, но это не произойдёт... Но подождите, есть объяснение: в отличие от вычисляемых свойств выполнение наблюдателей **откладывается до следующего цикла обновления**, который Vue использует для поиска изменений. Получается здесь происходит то, что `console.log` действительно вызывается, но после завершения теста.

Обратите внимание, что мы изменяем `inputValue` *непосредственным* образом, обращаясь к свойству `vm`. Если мы изменяем свойство данных подобным образом, нам нужно использовать функцию `vm.$nextTick`[36], чтобы отложить выполнение кода до следующего цикла обновления:

[36] https://ru.vuejs.org/v2/api/#vm-nextTick

```
it('вызывается с новым значением в других случаях', done => {
  cmp.vm.inputValue = 'foo'
  cmp.vm.$nextTick(() => {
    expect(spy).toBeCalled();
    done();
  });
});
```

В таком случае необходим вызов функции `done`*, которую мы получаем в качестве параметра. Это односторонний способ Jest*[37] *для проверки асинхронного кода.*

Однако есть **гораздо лучший способ**. Методы, предоставляемые Vue Test Utils, такие как `emitted` или `setData`, сами заботятся об этом, упрощая тем самым тестирование вычисляемых свойств.

```
it('вызывается с новым значением в других случаях', () => {
  cmp.setData({ inputValue: 'foo' });
  expect(spy).toBeCalled();
});
```

Мы можем применить ту же стратегию для следующего теста, с той разницей, что шпион не следует вызывать:

```
it('не вызывается, если значение пустое (с удалением пробелов)', next => {
  cmp.setData({ inputValue: '    ' });
  expect(spy).not.toBeCalled();
});
```

Наконец, написание теста, где шпион `console.log` в методе-наблюдателе не должен вызываться при одинаковых значений, будет чуть сложнее. Внутреннее состояние по умолчанию пустое, поэтому сначала нам нужно его изменить, дождаться следующего тика, затем очистить mock-объект, чтобы сбросить количество вызовов, и изменить `inputValue` снова. Затем, после второго тика, мы можем проверить шпиона и закончить тест. Опять же, благодаря методам Vue Test Utils нам не требуется использовать `$nextTick`.

Это может быть проще, если мы пересоздадим компонент в самом начале, переопределив свойство `data`. Помните, что вы можете переопределить любую опцию компонента, используя второй параметр функций `mount` или `shallowMount`:

[37] *https://jestjs.io/docs/en/asynchronous.html*

```
it('не вызывается, если значение одно и то же', () => {
  cmp = shallowMount(Form, {
    data: () => ({ inputValue: 'foo' })
  });
  cmp.setData({ inputValue: 'foo' });
  expect(spy).not.toBeCalled();
});
```

Резюме

В этой главе вы узнали, как тестировать часть логики компонентов Vue: вычисляемые свойства и наблюдатели. Мы разобрали различные тестовые сценарии, с которыми вы можете столкнуться, проверяя их. Вероятно, вы также изучили некоторые внутренние функции Vue, такие как циклы обновления `nextTick`.

Код этой статьи можно найти в этом репозитории[38].

[38] https://github.com/alexjoverm/vue-testing-series/tree/Test-State-Computed-Properties-and-Methods-in-Vue-js-Components-with-Jest

Тестирование методов и имитация зависимостей

В этой главе узнаем, как тестировать методы и справляться с имитацией зависимостями модулей.

Что мы должны тестировать в методах? Это вопрос мы задали себе, когда мы начали выполнять модульные тесты. Все сводится к тому, чтобы **проверить, что делает этот метод, и только это**. Это означает, что нам нужно **избегать вызовов любой зависимости**, поэтому нам нужно их имитировать (mock).

Давайте добавим событие `onSubmit` в форме компонента `Form.vue`, который мы создали в предыдущей главе:

```
<form @submit.prevent="onSubmit(inputValue)">
```

Модификатор `.prevent` — это просто удобный способ вызвать `event.preventDefault()`, чтобы не перезагружать страницу. Теперь внесите некоторые изменения, чтобы сделать запрос к API в метод `onSubmit` и сохранить результат с данными в массив `results`:

```
export default {
  data: () => ({
    inputValue: '',
    results: []
  }),
  methods: {
    onSubmit(value) {
      axios
        .get('https://jsonplaceholder.typicode.com/posts?q=' + value
        .then(results => {
          this.results = results.data;
        });
    }
  }
};
```

Метод использует axios для выполнения HTTP-вызова конечной точки «posts» http://jsonplaceholder.typicode.com, которая является всего лишь RESTful API для такого рода примеров, с параметром запроса `q` мы можем искать посты, используя предоставленный `value` в виде параметра.

Для тестирования метода `onSubmit`:

- Мы не хотим вызывать фактический метод `axios.get`
- Мы хотим проверить, что он вызывает axios (но не настоящие), и он возвращает промис
- Этот колбэк промиса должен установить `this.results` результат промиса

Это, вероятно, одно из самое тяжёлое в тестировании, когда у вас есть внешние зависимости, а также те промисы, которые делают что-то внутри. Что нам нужно сделать, это **имитировать внешние зависимости**.

Имитация зависимостей внешних модулей

Jest предоставляет действительно отличную систему, которая позволяет вам имитировать все довольно удобным способом. Для этого вам не нужны дополнительные библиотеки. Мы уже видели `jest.spyOn` и `jest.fn` для создания шпионов и создания функций-заглушек, хотя этого недостаточно для данного случая.

Нам нужно имитировать весь модуль `axios`. Вот тут `jest.mock` выходит на сцену. Это позволяет нам легко имитировать зависимости модулей, написав в верхней части файла:

```
1 jest.mock('dependency-path', implementationFunction);
```

Вам следует знать, что **`jest.mock` поднят**, что означает, что он будет помещён наверху. Поэтому:

```
1 jest.mock('something', jest.fn);
2 import foo from 'bar';
3 // ...
```

Эквивалентно:

```
1 import foo from 'bar';
2 jest.mock('something', jest.fn); // это в конечном итоге превысит импорт и все
3 // ...
```

К дате написания я все ещё не видел много информации в Интернете о том, как сделать в Jest то, что мы собираемся делать сейчас. К счастью, вам не нужно проходить ту же борьбу.

Давайте напишем имитацию для axios в верхней части тестового файла `Form.test.js` и соответствующий тестовый пример:

```
jest.mock('axios', () => ({
  get: jest.fn()
}));

import { shallowMount } from '@vue/test-utils';
import Form from '../src/components/Form';
import axios from 'axios'; // axios здесь, но их имитация вверху!

// ...

it('Вызывает axios.get', () => {
  cmp.vm.onSubmit('an');
  expect(axios.get).toBeCalledWith(
    'https://jsonplaceholder.typicode.com/posts?q=an'
  );
});
```

Это здорово, мы на самом деле создали имитацию для axios, поэтому оригинальная библиотека axios не вызывает ни один HTTP-запрос. И мы даже проверяем с помощью `toBeCalledWith`, что он был вызван с правильными параметрами. Но мы все ещё что-то упустили: ***мы не проверяем, что он возвращает обещание***.

Сначала нам нужно сделать так, чтобы наш подстановочный метод `axios.get` возвращал промис. `jest.fn` принимает фабричную функцию в виде параметра, поэтому мы можем использовать её для определения реализации:

```
jest.mock('axios', () => ({
  get: jest.fn(() => Promise.resolve({ data: 3 }))
}));
```

Но тем не менее, мы не можем получить доступ к промису, потому что мы его не возвращаем. При тестировании хорошей практикой является возможность вернуть что-то из функции, когда это возможно, что значительно облегчает тестирование. Давайте сделаем это в методе `onSubmit` компонента `Form.vue`:

```
export default {
  methods: {
    // ...
    onSubmit(value) {
      const getPromise = axios.get(
        'https://jsonplaceholder.typicode.com/posts?q=' + value
      );

      getPromise.then(results => {
        this.results = results.data;
      });

      return getPromise;
    }
  }
};
```

Затем мы можем использовать предельно понятный синтаксис из ES2017 `async/await` в тесте для проверки результата промиса:

```
it('Вызывает axios.get и проверяет результат промиса', async () => {
  const result = await cmp.vm.onSubmit('an');

  expect(result).toEqual({ data: [3] });
  expect(cmp.vm.results).toEqual([3]);
  expect(axios.get).toBeCalledWith(
    'https://jsonplaceholder.typicode.com/posts?q=an'
  );
})
```

Вы можете видеть, что мы не только проверяем результат промиса, но также и то, что внутреннее состояние компонента `results` обновляется, как и ожидалось, путём выполнения `expect(cmp.vm.results).toEqual([3])`.

Вынесение mock-объектов во внешние файлы

Jest позволяет нам разделять все наши подстановочные объекты в отдельном JavaScript-файле, помещая их в папку `__mocks__`, чтобы тесты были максимально чистыми и понятными.

Поэтому мы можем взять блок `jest.mock...` из файла `Form.test.js` и перенести в собственный файл:

```
// test/__mocks__/axios.js
module.exports = {
  get: jest.fn(() => Promise.resolve({ data: [3] }))
};
```

Точно так же, без каких-либо дополнительных усилий, Jest автоматически применяет подстановочный объект во всех наших тестах, поэтому нам не нужно делать что-либо лишнее или изменять тесты. Обратите внимание, что имя модуля должно совпадать с именем файла. Если вы снова запустите тесты, они все равно должны пройти.

Имейте в виду, что реестр модулей и состояние подстановочного объекта сохранены, поэтому, если вы впоследствии напишите ещё один тест, вы можете получить нежелательные результаты:

```
it('Вызывает axios.get', async () => {
  const result = await cmp.vm.onSubmit('an');

  expect(result).toEqual({ data: [3] });
  expect(cmp.vm.results).toEqual([3]);
  expect(axios.get).toBeCalledWith(
    'https://jsonplaceholder.typicode.com/posts?q=an'
  );
});

it('Axios не должен быть вызван в данном случае', () => {
  expect(axios.get).toBeCalledWith(
    'https://jsonplaceholder.typicode.com/posts?q=an'
  );
});
```

Второй тест должен потерпеть неудачу, но это не так! Это потому, что `axios.get` был вызван на тест раньше.

По этой причине, это хорошая практика, чтобы очистить реестр модулей и все подстановочные объекты, поскольку ими управляет Jest для имитации их реализации. Для этого вы можете добавить в свой `beforeEach`:

```
beforeEach(() => {
  cmp = shallowMount(Form);
  jest.resetModules();
  jest.clearAllMocks();
});
```

Теперь каждый тест будет начинаться с чистых подстановочных объектов (mocks) и модулей, как это и должно быть в модульном тестировании.

Резюме

Функционал имитации Jest, наряду с моментальным снимком, — это то, что я люблю больше всего в Jest! С его помощью очень легко протестировать то, что обычно довольно сложно поддаётся тестированию, а также сосредоточиться на написании более быстрых и лучших изолированных тестов и сохранить свою кодовую базу пуленепробиваемой.

Код этой главы можно найти в этом репозитории[39].

[39] https://github.com/alexjoverm/vue-testing-series/tree/Test-State-Computed-Properties-and-Methods-in-Vue-js-Components-with-Jest

Тестирование слотов Vue.js в Jest

В этой главе вы узнаете, как тестировать контент, распространяемый с помощью слотов и именованных слотов.

Слоты — это способ распространения контента в мире веб-компонентов. Слоты Vue.js создаются по спецификациям веб-компонентов[40], что означает, что если вы узнаете, как их использовать во Vue.js, то это знание вам будет полезным для вас в будущем ;).

Они создают структуру компонентов более гибкой, перенося ответственность за управление состоянием на родительский компонент. Например, у нас может быть компонент `List` и другого рода компоненты элементов, такие как `ListItem` и `ListItemImage`. Они будут использоваться как показано ниже:

```
<template>
  <List>
    <ListItem :someProp="someValue" />
    <ListItem :someProp="someValue" />
    <ListItemImage :image="imageUrl" :someProp="someValue" />
  </List>
</template>
```

Внутреннее содержимое `List` — это сам слот, доступный через тег `<slot>`. Таким образом, реализация `List` выглядит так:

```
<template>
  <ul>
    <!-- слот будет равен тому, что внутри <List> -->
    <slot></slot>
  </ul>
</template>
```

И, скажем, что элемент `ListItem` выглядит так:

```
<template>
  <li>{{ someProp }}</li>
</template>
```

Тогда окончательный результат, отрисованный Vue.js, будет следующим:

[40] https://github.com/w3c/webcomponents/blob/gh-pages/proposals/Slots-Proposal.md

```
<ul>
  <li>someValue</li>
  <li>someValue</li>
  <li>someValue</li> <!-- предположим, что реализация одна и та же для ListItemImage\
 -->
</ul>
```

Создание MessageList на основе слотов

Давайте посмотрим на компонент `MessageList.vue`:

```
<template>
  <ul>
    <Message
      @message-clicked="handleMessageClick"
      :message="message"
      v-for="message in messages"
      :key="message"
    />
  </ul>
</template>
```

Компонент `MessageList` имеет «жёстко закодированный» компонент `Message` внутри. В некотором смысле это более автоматизировано, но в другом случае такой компонент не является гибким. Что делать, если вы хотите иметь разные типы компонентов `Message`? Как насчёт изменения его структуры или стилизации? Вот где слоты пригодятся.

Давайте будем использовать `Message.vue` для использования слотов. Сначала переместите эту часть `<Message...` в компонент `App.vue`, вместе с методом `handleMessageClick`, потому что он использовался извне:

```
<template>
  <div id="app">
    <MessageList>
      <Message
        @message-clicked="handleMessageClick"
        :message="message"
        v-for="message in messages"
        :key="message"/>
    </MessageList>
  </div>
```

```
</template>

<script>
import MessageList from './components/MessageList';
import Message from './components/Message';

export default {
  name: 'app',
  data: () => ({ messages: ['Привет, Джон', 'Как дела, Пако?'] }),
  methods: {
    handleMessageClick(message) {
      console.log(message);
    }
  },
  components: {
    MessageList,
    Message
  }
};
</script>
```

Не забудьте импортировать компонент `Message` и добавить его в опцию `components` в `App.vue`.

Затем в `MessageList.vue` мы можем удалить ссылки на `Message`, в итоге он будет выглядеть так:

```
<template>
  <ul class="list-messages">
    <slot></slot>
  </ul>
</template>

<script>
export default {
  name: 'MessageList'
};
</script>
```

Свойства экземпляра `$children` и `$slots`

Vue-компоненты имеют две свойства экземпляра, которые полезны для доступа к слотам:

- `$children`: массив экземпляров Vue-компонента в слоте по умолчанию.
- `$slots`: объект VNodes, содержащий все слоты, определённые в экземпляре компонента.

Объект `$slots` имеет больше доступных данных. Фактически, `$children` — это всего лишь часть переменной `$slots`, к которой можно получить доступ таким же образом путём доступа к `$slots.default`, отфильтрованного экземплярами Vue-компонента:

```
const children = this.$slots.default
  .map(vnode => vnode.componentInstance)
  .filter(cmp => !!cmp)
```

Тестирование слотов

Вероятно, мы хотим протестировать большую часть слотов, когда они попадают в компонент, и для этого мы можем повторно использовать навыки, полученные из третьей главы.

В настоящее время большинство тестов в `MessageList.test.js` не пройдут, поэтому давайте удалим их все (или закомментируем их) и сосредоточимся на тестировании слотов.

Единственное, что мы можем проверить, — убедиться, что компоненты `Message` находятся в элементе `ul` с классом `list-messages`. Для передачи слотов компоненту `MessageList`, мы можем использовать свойство `slots` объекта опций у методов `mount` или `shallowMount`. Итак, давайте создадим метод `beforeEach`[41] со следующим кодом:

```
beforeEach(() => {
  cmp = shallowMount(MessageList, {
    slots: {
      default: '<div class="fake-msg"></div>'
    }
  });
});
```

Поскольку мы просто хотим проверить, отрисовываются ли сообщения, мы можем искать `<div class="fake-msg"></div>` следующим образом:

```
it('Сообщения добавлены в элемент ul.list-messages', () => {
  const list = cmp.find('ul.list-messages');
  expect(list.findAll('.fake-msg').length).toBe(1);
})
```

И это должно пройти успешно. Опция `slots` также принимает объявление компонента и даже массив, поэтому мы могли бы написать:

[41] https://jestjs.io/docs/en/api.html#beforeeachfn-timeout

```
import AnyComponent from 'anycomponent';
// ...
shallowMount(MessageList, {
  slots: {
    default: AnyComponent // Или [AnyComponent, AnyComponent]
  }
});
```

Проблема с этой опцией заключается в том, что она очень ограничена, например, вы не можете переопределять входные параметры, как в данном случае нам нужно это для компонента `Message`, так как он имеет обязательное свойство. Это должно повлиять на случаи, когда вам на самом деле нужны для тестирования слотов с ожидаемыми компонентами. Например, если вы хотите удостовериться, что `MessageList` ожидает только `Message` в качестве слотов. Такой возможности во Vue Test Utils нет, вы можете почитать обсуждение в соответствующей ишью[42]: можно передать только компонент, но не экземпляр.

В качестве обходного решения мы можем воспользоваться render-функцией[43]. Поэтому мы можем переписать тест, чтобы быть более точным:

```
beforeEach(() => {
  const messageWrapper = {
    render(h) {
      return h(Message, { props: { message: 'hey' } })
    }
  };

  cmp = shallowMount(MessageList, {
    slots: {
      default: messageWrapper
    }
  });
});

it('Сообщения добавлены в элементе ul.list-messages', () => {
  const list = cmp.find(MessageList)
  expect(list.find(Message).isVueInstance()).toBe(true)
});
```

[42] https://github.com/vuejs/vue-test-utils/issues/41#issue-255235880
[43] https://ru.vuejs.org/v2/guide/render-function.html

Тестирование именованных слотов

Неименованный слот, который мы использовали выше, называется *слотом по умолчанию*, но у нас может быть несколько слотов, используя именованный слот. Давайте добавим заголовок к компоненту `MessageList.vue`:

```
<template>
  <div>
    <header class="list-header">
      <slot name="header">
        Это заголовок по умолчанию
      </slot>
    </header>
    <ul class="list-messages">
      <slot></slot>
    </ul>
  </div>
</template>
```

Используя `<slot name="header">`, мы определяем ещё один слот для заголовка. Вы можете увидеть текст `Это заголовок по умолчанию` внутри слота, который отображается как контент по умолчанию, когда слот не передаётся компоненту, это также относится к слоту по умолчанию.

Затем из `App.vue` мы можем использовать добавление заголовка к компоненту `MessageList` с помощью атрибута `slot="header"`:

```
<template>
  <div id="app">
    <MessageList>
      <header slot="header">
        Потрясающий заголовок
      </header>
      <Message
        @message-clicked="handleMessageClick"
        :message="message"
        v-for="message in messages"
        :key="message"/>
    </MessageList>
  </div>
</template>
```

Пришло время написать для него модульный тест. Тестирование именованных слотов точно такое же, как тестирование слота по умолчанию, тот же самый процесс. Таким образом, мы можем начать с тестирования того, что слот заголовка отображается в элементе `<header class="list-header">`, и он отрисовывает текст по умолчанию, когда ни один из слотов заголовка не передаётся. В `MessageList.test.js`:

```
it('Слот заголовка отрисовывает текст заголовка по умолчанию', () => {
  const header = cmp.find('.list-header')
  expect(header.text().trim()).toBe('Это заголовок по умолчанию')
})
```

Затем делаем то же самое, но проверка содержимого по умолчанию заменяется, когда мы имитируем слот заголовка:

```
it('Слот заголовка отрисовывается внутри .list-header', () => {
  const component = shallowMount(MessageList, {
    slots: {
      header: '<div>Какой удивительный заголовок</div>'
    }
  })

  const header = component.find('.list-header')
  expect(header.text().trim()).toBe('Какой удивительный заголовок')
})
```

Посмотрите, что слот заголовка, используемый в этом последнем тесте, завернут в `<div>`. Важно, чтобы слоты были обёрнуты тегом HTML, иначе Vue Test Utils будет жаловаться.

Тестирование спецификаций слота

Мы тестируем, как и где отрисовываются слоты, и, вероятно, это то, что нам в большинстве случаев нужно. Однако это ещё не все. Если вы передаёте экземпляры компонентов в качестве слотов, по аналогии как в слоте по умолчанию с `Message`, вы можете протестировать связанную с ним функциональность.

Будьте осторожны с тем, что вы тестируете здесь, это, вероятно, то, что вам не нужно делать в большинстве случаев, поскольку функциональные тесты компонента должны принадлежать этому тесту компонента. Когда мы говорим о тестировании функциональности слотов, мы тестируем, как слот должен вести себя **в контексте компонента, в котором этот слот используется**, и это не очень распространено. Обычно мы просто передаём слот и забываем об нём. Поэтому не следует придерживаться следующего примера, единственная его цель — продемонстрировать, как работает инструмент.

Предположим, что по какой-либо причине в контексте компонента `MessageList` все компоненты `Message` должны иметь длину больше 5 символов во входном параметре `message`. Мы можем проверить это таким образом:

```
it('Длина сообщения превышает 5 символов', () => {
  const messages = cmp.findAll(Message)
  messages.wrappers.forEach(c => {
    expect(c.vm.message.length).toBeGreaterThan(5)
  })
})
```

Метод `findAll` возвращает объект, содержащий массив `wrappers`, где мы можем получить доступ к свойству экземпляра компонента `vm`. Этот тест завершится неудачно, потому что сообщение имеет длину 3, поэтому перейдите к функции `beforeEach` и увеличьте длину сообщение:

```
beforeEach(() => {
  const messageWrapper = {
    render(h) {
      return h(Message, { props: { message: 'привет' } });
    }
  };
// ...
```

Затем этот тест должен пройти.

Резюме

Тестирование слотов очень просто, обычно мы хотели бы проверить, что они размещены и отрисованы так, как мы хотим, так же, как при тестировании стиля и структуры, зная, как ведут себя слоты или могут быть имитированы. Вероятно, вам не понадобиться часто тестировать функциональность слота. Имейте в виду, проверять только то, что относится непосредственно к слотам при тестировании слотов, и дважды подумайте над тем, что вы тестируете, должно ли это находится в тесте самого компонента или слота.

Код этой главы можно найти в этом репозитории[44].

[44] https://github.com/alexjoverm/vue-testing-series/tree/test-slots

Улучшение конфигурации Jest с помощью псевдонимов модулей

В этой главе вы узнаете, как использовать псевдонимы модуля Jest, чтобы избежать использования относительных путей.

Менеджеры модулей, которые у нас есть в сообществе JavaScript, главным образом ES-модули и CommonJS, не поддерживают пути на основе проектов. Они поддерживают только относительные пути для наших собственных модулей и пути для каталога `node_modules`. Когда проект немного растёт, обычно встречаются такие пути:

```
1 import SomeComponent from '../../../../components/SomeComponent';
```

К счастью, у нас есть разные способы справиться с этим, таким образом мы могли определить псевдонимы для каталог относительно корня проекта, поэтому мы могли бы использовать написать приведённую выше строку так, как показано ниже:

```
1 import SomeComponent from '@/components/SomeComponent';
```

Знак `@` здесь является произвольным символом для определения корневого проекта, вы можете определить свой собственный. Давайте посмотрим, какие решения мы можем применить для псевдонимов модулей. Начнём с того места, где мы остановились в предыдущей главе[45].

Псевдонимы Webpack

Псевдонимы Webpack[46] очень просты в настройке. Вам просто нужно добавить свойство `resolve.alias` в конфигурацию вашего webpack. Если вы посмотрите на `build/webpack.base.conf.js`, там уже определён один псевдоним:

[45] https://github.com/alexjoverm/vue-testing-series/tree/test-slots
[46] https://webpack.js.org/configuration/resolve/#resolve-alias

```
module.exports = {
  // ...
  resolve: {
    extensions: ['.js', '.vue', '.json'],
    alias: {
      'vue$': 'vue/dist/vue.esm.js',
    }
  }
};
```

Принимая это как точку входа, мы можем добавить простой псевдоним, указывающий на папку `src`, и использовать его в качестве корня проекта:

```
module.exports = {
  // ...
  resolve: {
    extensions: ['.js', '.vue', '.json'],
    alias: {
      'vue$': 'vue/dist/vue.esm.js',
      '@': path.join(__dirname, '..', 'src')
    }
  }
};
```

Теперь благодаря этому мы можем получить доступ к чему-либо, взяв корневой проект как символ `@`. Перейдём к `src/App.vue` и изменим импорт на эти два компонента:

```
import MessageList from '@/components/MessageList';
import Message from '@/components/Message';
// ...
```

И если мы запустим `npm start` и откроем браузер по URL-адресу `localhost: 8080`, это должно работать из коробки.

Однако, если мы попытаемся выполнить тесты, запустив `npm t`, мы увидим, что Jest не находит модули. Мы все ещё не настроили Jest для работы псевдонимов модулей. Итак, перейдём к `package.json`, где находится конфигурация Jest, и добавим `"@/([^\\.]*)$": "<rootDir>/src/$1"` в `moduleNameMapper`:

```
{
  "jest": {
    "moduleNameMapper": {
      "@(.*)$": "<rootDir>/src/$1",
      "^vue$": "vue/dist/vue.common.js"
    }
  }
}
```

Давайте объясним это:

- `@(.*)$`: Что бы ни начиналось с символа `@` и продолжалось в буквальном смысле (`(.*)$`) до конца строки, группируя путь, используя скобки
- `<rootDir>/src/$1`: `<rootDir>` — специальное слово Jest, которое означает корневой каталог. Затем мы сопоставляем его с `src`, а с помощью `$1` мы добавляем любую составную часть из регулярного выражения `(.*)`.

Например, `@/components/MessageList` будет ссылаться на `../src/components/MessageList`, когда вы импортируете его из каталогов `src` или `test`.

Это действительно так. Теперь вы можете даже обновить файл `App.test.js`, чтобы использовать псевдоним, поскольку теперь он может использоваться в тестах:

```
import { shallowMount } from '@vue/test-utils'
import App from "@/App"
// ...
```

И это будет работать для файлов `.vue` и `.js`.

Несколько псевдонимов

Очень часто для удобства используются несколько псевдонимов, поэтому вместо использования только `@` для определения корневого каталога вы можете использовать ещё многие другие. Например, предположим, что у вас есть каталог `actions` и `models`. Если вы создаёте псевдоним для каждого из них, а затем перемещаете каталоги, то вам просто нужно изменить псевдонимы вместо обновления всех выражений импорта на него по всей кодовой базе. Это мощь псевдонимов модулей, они делают вашу кодовую базу более удобной и чистой.

Давайте добавим псевдоним `components` в `build/webpack.base.conf.js`:

```
module.exports = {
  // ...
  resolve: {
    extensions: ['.js', '.vue', '.json'],
    alias: {
      'vue$': 'vue/dist/vue.esm.js',
      '@': path.join(__dirname, '..', 'src')
      'components': path.join(__dirname, '..', 'src', 'components')
    }
  }
};
```

Затем нам просто нужно добавить его также в конфигурацию Jest в `package.json`:

```
{
  "jest": {
    "moduleNameMapper": {
      "@(.*)$": "<rootDir>/src/$1",
      "components(.*)$": "<rootDir>/src/components/$1",
      "^vue$": "vue/dist/vue.common.js"
    }
  }
}
```

Вот так просто. Теперь мы можем попробовать в `App.vue` использовать обе формы:

```
import MessageList from 'components/MessageList';
import Message from '@/components/Message';
```

Остановите и перезапустите выполнение тестов, и это должно работать, а также если вы запустите `npm start` и попробуйте, что из этого выйдет.

Другие решения

Я видел babel-plugin-webpack-alias[47], специально используемый для других фреймворков тестирования, таких как mocha[48], который не имеет модуля-преобразователя (module mapper).

Я сам не пробовал, потому что с использованием Jest это не нужно, но если вы хотите попробовать, пожалуйста, поделитесь результатом!

[47] https://github.com/trayio/babel-plugin-webpack-alias
[48] https://mochajs.org/

Вывод

Добавление псевдонимов модулей очень просто и может держать вашу кодовую базу намного чище и проще в обслуживании. Jest упрощает их определение, вам просто нужно поддерживать синхронизацию с псевдонимами Webpack, и вы можете прощаться с адом относительных путей.

Полный пример вы можете найти на GitHub[49].

[49] https://github.com/alexjoverm/vue-testing-series/tree/Enhance-Jest-configuration-with-Module-Aliases